5 Pilhas

" Ao contrario do que se pode pensar, o conceito de algoritmo não foi criado para satisfazer as necessidades da computação. Pelo contrário, a programação de computadores é apenas um dos campos da aplicação dos algoritmos "

- **Saliba** -

Sumário:

5.1 – Conceitos

Uma **pilha (stack)** é uma lista linear onde as operações de inserção, remoção e consulta, são feitas numa extreminada denomina-se por **topo** ou lado aberto.

Como o último elemento que entrar na pilha é o primeiro a sair, esta estrutura é conhecida como uma lista do tipo **LIFO**. Este termo deriva da frase *"last-in", "first-out".* Vejamos alguns exemplos:

As pilhas são estruturas de dados que são muito utilizadas em aplicações de software. Por exemplo, na execução de programas, a pilha é utilizada para a chamada de procedimentos e funções, onde armazena o valor de retorno e os argumentos. A medida que novos procedimentos ou novas funções são chamadas, os parâmetros e o valor de retorno serão empilhados. Estes, serão desempilhados quando esses procedimentos ou funções chegarem ao seu término.

No desenvolvimento de editores de texto, pilha é utilizada para activar a funcionalidade "Undo" que permite cancelar as operações recentes e reverter o documento ao estado anterior a essa operação.

Os navegadores de internet também armazenam os endereços mais recentes numa pilha. Todas as vezes que um novo site é consultado, o seu endereço é armazenado numa pilha de endereços. Se utilizar-nos a operação "back", o navegador permite que o utilizador retorne ao último site consultado, retirando o seu endereço da pilha.

As pilhas podem ser implementadas na organização estática (utilizando vectores) ou dinâmica (utilizando ponteiros) sem diferenças significativas em termos de eficiência, uma vez que a estrutura só admite estas operações no topo da estrutura.

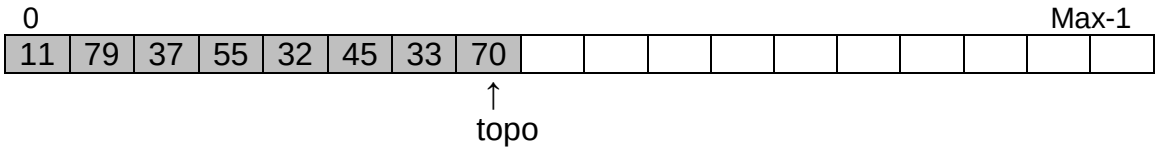
5.2- Tipo Abstracto de Dados

Um possível conjunto de operações para criar um tipo abstracto de dados pilha é descrito pelas operações:

Inicializar um pilha;
Verificar se a pilha está vazia;
Verificar se a pilha está cheia;
Inserir um elemento no topo da pilha "push down";
Remover um elemento do topo da pilha " pop-up" ;
Consultar um elemento na pilha;
Número de Elementos de uma pilha;

5.3- Pilha Estática

Uma forma muito simples de implementar esta estrutura, consiste em fixar o ínicio da pilha na primeira posição do vector. Com essa estratégia, as inserções são feitas por um contador denominado por topo, que é incrementado em uma unidade e, as remoções são feitas pelo mesmo contador que é decrementado em uma unidade.



5.3.1- Estrutura de Dados

```
// Códigos de Erro

#define NOT_FOUND      -1    // Item não existe
#define  OK             0    // Operação realizada com sucesso
#define  STACK_FULL     1    //  Pilha  cheia
#define  STACK_EMPTY    2    //  Pilha vazia

#define TAM  100             // Tamanho do vector

typedef struct               // Estrutura de dados
{
    int  chave;
    float valor;
}TItem;

typedef struct
{
   Titem pilha[TAM];
    int Topo;
} TPilha;
```

5.3.2- Implementação do Interface

```
/* ---------------------------------------------------------------------------------------------
Especificação do Interface : pilhaSequencial.h
Objectivo: Disponibilizar as operações sobre uma pilha sequencial
---------------------------------------------------------------------------------------------*/
#ifndef  PILHASEQ_H_INCLUDED
#define PILHASEQ_H_INCLUDED

//------------------------------------------------------ Definição dos códigos de erro

#define NOT_FOUND      -1    // Item não existe
#define  OK             0    // Operação realizada com sucesso
```

```
#define  STACK_FULL      1      //  Pilha  cheia
#define  STACK_EMPTY     2      //  Pilha vazia

#define TAM  100                //  Tamanho do vector

//------------------------------------------------------- Tipo de Dados a ser exportado
typedef struct  pilha  TPilha;

//------------------------------------------------------- Protótipos das Funções
// Funções exportadas
void InicializarPilha ( TPilha  *pilha );
//  inicializa uma Pilha Sequencial

boolean vaziaPilha ( TPilha  pilha );
// verifica se a Pilha Sequencial está vazia

boolean cheiaPilha (TPilha  pilha);
// verifica se a Pilha Sequencial está cheia

int consultaPilha ( TPilha pilha, int x );
//  Devolve o conteudo do elemento que está no topo da pilha

int tamanhoPilha ( Tpilha pilha );
//  Devolve o número de elementos inserido numa pilha

#endif // PILHASEQ_H_INCLUDED
```

5.3.3- Implementação das Operações

Agora, vamos implementar as estratégias utilizadas nos algoritmos que descrevem o funcionamento das funções descritas no arquivo interface e a função que controla os erros das operações.

De forma analoga a lista sequencial, inicializar uma pilha, consiste em associar o topo à um valor anterior ao primeiro elemento do vector que a contêm.

```
/*-----------------------------------------------------------------------------------------------------
Objectivo: Criar  uma estrutura de dados do tipo pilha
Parâmetro Entrada: pilha com lixo residual
Parâmetro de Saída: Pilha inicializada
-----------------------------------------------------------------------------------------------------*/
void  inicializarPilha ( TPilha  *pilha )
{
   pilha->topo = -1;
}
```

A operação para determinar se uma pilha está vazia, consiste em verificar se ela não possui elementos. Pelo operador anterior, basta verificar se o topo faz referência à uma posição anterior ao primeiro elemento do vector.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Verificar se a pilha está vazia
Parâmetro Entrada: Pilha
Retorno da Função: Verdadeiro  ou Falso
-------------------------------------------------------------------------------------------------------*/
Boolean vaziaPilha ( TPilha  pilha )
{
    return  ( pilha.topo == -1 );
}
```

A operação para determinar se uma pilha está cheia, consiste em verificar se todos os elementos do vector estão preenchidos. Para isso, basta comparar o topo com o índice do último elemento vector.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Verificar se a pilha está cheia
Parâmetro Entrada: Pilha
Retorno da Função: Verdadeiro  ou Falso
-------------------------------------------------------------------------------------------------------*/
Boolean cheiaPilha ( TPilha  pilha )
{
   return ( pilha.topo == TAMANHO-1 );
}
```

A operação para inserir um elemento numa pilha, denominada por empilhar, consiste em verificar em primeiro lugar, se a pilha não está cheia. Se essa condição for verdadeira, o processo de inserção consiste em deslocar o topo para uma posição à direita (adicionar uma unidade) e em seguida inserir o elemento nessa posição.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Inserir um elemento no topo da pilha
Parâmetro Entrada: Um elemento qualquer e uma pilha
Parâmetro de Saída: Pilha actualizada
Retorno da Função: Código de erro (QUEUE_FULL ou OK )
-------------------------------------------------------------------------------------------------------*/
int  empilhar ( int x, TPilha *pilha )
{
    if ( CheiaPilha (pilha) )
       return STACK_FULL;
    pilha->topo ++;
    pilha->item[pilha->topo] =  x;
    return OK;
}
```

A operação para remover de um elemento numa pilha, denominada por desempilhar, consiste em verificar em primeiro lugar se a pilha não está vazia. Se essa condição for verdadeira, o processo de remoção consiste em guardar numa variável de memória o elemento referenciado pelo topo e em seguida, deslocar o topo para uma posição à esquerda (subtrair uma unidade).

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Remover um  elemento que está no topo da pilha
Parâmero Entrada: Pilha
Parâmetro de Saída: Pilha actualizada e a informação removida
Retorno da Função: Código de erro ( STACK_EMPTY ou OK).
-------------------------------------------------------------------------------------------------*/
int  desempilhar ( TPilha *pilha, int *x )
{
    if ( VaziaPilha (pilha) )
        return STACK_EMPTY;
   *x = pilha->item[pilha->topo];
   pilha->topo--;
   return OK;
}
```

A operação para consultar ou alterar um elemento de uma pilha consiste em verificar em primeiro lugar se a pilha está vazia. Se essa condição for verdadeira, devolver o correspondente código de erro. No caso contrário, devolver o conteudo do elemento que está no topo da pilha.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Consultar um elemento da pilha
Parâmetro Entrada: Pilha
Parâmetro de Saída: Conteudo da Pilha
Retorno da Função: Código de erro (STACK_EMPTY ou OK)
-------------------------------------------------------------------------------------------------*/
int  acessoPilha ( TPilha pilha, int *x )
{
    if ( vaziaPilha (pilha) )
      return STACK_EMPTY;
    *x = pilha->item[pilha->topo];
    return OK;
}
```

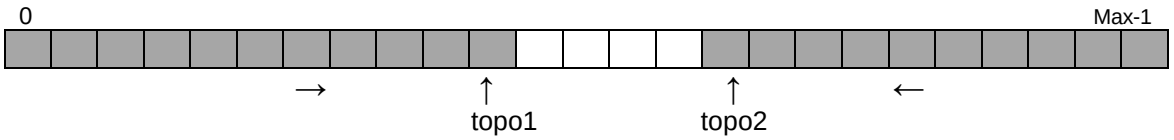
Devemos salientar que não podemos aceder de forma directa ao conteudo de um elemento no interior da pilha. Para realizar essa operação é necessário remover todos os elementos que vão desde o topo até esse elemento. Consultar a informação e voltar a colocar os elementos removidas na ordem que foram inseridos.

A operação para determinar o número de elementos inseridos numa pilha é muito simples. Ela é determinada pelo posição do topo mais uma unidade.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Determinar o número de elementos inseridos numa pilha
Parâmetro Entrada: Pilha
Retorno da Função: Número de elementos
-------------------------------------------------------------------------------------------------*/
int tamanhoPilha(TPilha  pilha)
{
   return ( pilha.topo + 1 );
}
```

5.4 - Duas Pilha num Vector

Duas pilhas com implementação estática podem compartilhar o mesmo vector de forma eficiente se as pilhas forem posicionadas nas extremidades do vector e crescerem em direção opostas, em termos gráficos:



Esta representação pode ser declarada pela seguinte estrutura

```
typedef struct
{
   int topo1;
   int topo2;
   TItem pilha[MAX];
} TPDupla;
```

Vamos estudar em seguida, as operações que determinam um tipo abstracto de dados PilhaDupla. Devido a analogia com a pilha simples, omitiremos a descrição dos algoritmos que manipulam essa estrutura e, passamos a apresentar as correspondentes funções na linguagen C.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Criar  uma estrutura de dados do tipo pilha dupla
Parâmetro Entrada: Vector com lixo residual
Parâmetro de Saída: Pilha dupla inicializada
-------------------------------------------------------------------------------------------------*/
void inicializarPilhaDupla ( TPDupla *pilha )
{
   pilha->topo1 = -1;
   pilha->topo2 = MAX+1;
}
```

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Determinar o número de elementos inseridos numa pilha
Parâmetro Entrada: Vector com as pilhas, número da pilha
Retorno da Função: Número de elementos dessa pilha
-------------------------------------------------------------------------------------------------------*/
int tamanhoPilhaDupla ( TPDupla pilha, int np )
{
    if( np == 1 ) return(pilha.topo1 + 1);
    return(MAX+1- pilha.topo2);
}


/*-------------------------------------------------------------------------------------------------------
Objectivo: Verificar se uma pilha está vazia
Parâmetro Entrada: Vector com as pilhas e o número da pilha
Retorno da Função: Verdadeiro ou falso
-------------------------------------------------------------------------------------------------------*/
Boolean pilhaDuplaVazia ( TPDupla pilha, int np )
{
  if ( np == 1 )
     return ( pilha.topo1 == -1 );
  return ( pilha.topo2 == MAX+1 );
}

/*-------------------------------------------------------------------------------------------------------
Objectivo: Verificar se a  pilha está cheia
Parâmetro Entrada: Vector com as pilhas
Retorno da Função: Verdadeiro ou falso
-------------------------------------------------------------------------------------------------------*/
Boolean  pilhaDuplaCheia ( TPDupla pilha )
{
    return ( pilha.topo1 == (pilha.topo2 - 1) ) return;
}

/*-------------------------------------------------------------------------------------------------------
Objectivo: Inserir um elemento numa pilha
Parâmetro Entrada: Vector com as pilhas, elemento a inserir, número da pilha
Parâmetro de saída : Pilha actualizada
Retorno da Função: Código de erro (STACK_EMPTY ou OK)
-------------------------------------------------------------------------------------------------------*/
int empilhar ( TPDupla *pilha, TItem x, int np )
{
   if ( pilhaDuplaCheia (pilha) )
       return STACK_EMPTY;
   if (np == 1)
     {
        pilha->topo1++;
        pilha ->item[pilha->topo1] = x;
      }
   else
```

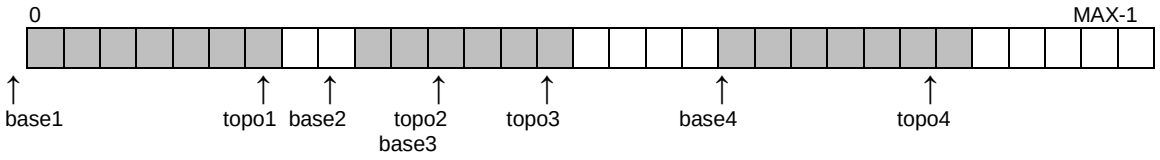
```
        {
          pilha->topo2--;
          pilha ->item[pilha->topo2] = x;
         }
    return OK;
}
```

Para consolidar a matéria, deixamos como exercício a operação para remover (desempilhar) um elemento numa determinada pilha.

5.5- Múltiplas Pilhas num Vector

Se tivermos mais do que duas pilhas a compatilharem o mesmo vector, a solução apresentada na secção anterior, não permite controlar as localizações de memória de cada pilha. Um vector tem apenas dois pontos fixos, o primeiro e o último elemento e, a solução que pretendemos deve controlar para cada pilha a sua base (o ponto onde inicia) e o seu topo (o ponto onde termina).

Para representarmos de forma eficiênte múltiplas pilhas num único vector, o topo de cada pilha deve fazer referência ao último elemento efectivo dessa pilha, enquanto a base de cada pilha deve fazer referência ao elemento anterior ao ínicio dessa pilha, em termos gráficos:



Esta representação pode ser declarada pela seguinte estrutura

```
typedef struct
{
   int base[NP+1];        // pilhas [0..NP-1] + pilha[NP] extraordinária
   int topo[NP+1];
   TItem pilha[MAX];
} TPMultipla;
```

onde NP é uma constante simbólica que representa o número de pilhas que pretendemos armazenar.

Com essa representação qualquer pilha k, para 0 ≤ k ≤ NP goza das seguintes propriedades:
a) A pilha[k] cresce da base[k]+1 até base[k+1];
b) A pilha[k] está vazia quando base[k] = topo[k];
c) A pilha[k] está cheia quando topo[k] = base[k+1].

Vamos estudar em seguida, algumas operações sobre este tipo abstracto de dados.

A operação de inicialização de uma pilha múltipla, consiste em igualar o topo de cada pilha a sua base, defenindo deste modo, uma pilha vazia. Para além disso, para evitar que as pilhas fiquem acumuladas numa parte do vector e com isso gerar uma grande movimentação de elementos com as primeiras inserçoes, as NP+1 pilhas são inicializadas com suas bases e topos distribuídas em intervalos aproximadamente iguais ao longo da estrutura.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Criar uma estrutura de pilha dupla
Parâmetro Entrada: Estrutura pilha dupla
Parâmetro de Saída: Estrutura pilha dupla inicializada
-------------------------------------------------------------------------------------------------------*/
void inicializarPilhaMultipla (TPMultipla *pilha)
{
  for( int i = 0; i <= NP ; i++ )
    {
      pilha->base[i] = ( i * (MAX / NP) ) - 1;
      p->topo[i] = p->base[i];
    }
}
```

A operação para verificar se uma pilha está cheia é muito simples. Mas, iremos tecer algumas considerações para justificar a existência de uma pilha extraordinária na declaração dessa estrutura.

Pela operação anterior, a pilha[0] contém o valor -1 e a pilha extraordinária, pilha[NP], contém de forma permanente o valor da última posição do vector mais uma unidade. Esses valores servem de sentinelas de início e fim do vector e nunca serão alterados pelas operações de inserção e de remoção de elementos.

Como os sentinelas, eles têm a finalidade de simplificar o teste de fim de pilha cheia, que consiste em comparar o topo de uma pilha com a base da pilha seguinte. Desse modo, evita-se tratar como caso especial, a última pilha real do vector.

Então, podemos concluir que a existência de uma pilha extraordinária na declaração da estrutura de pilha múltipla, tem por finalidade, simplificar as operações sobre essa estrutura, em especial a operação para verificar se uma pilha está cheia.

A operação para remover um elemento na k-ésima pilha, consiste em verificar em primeiro lugar, se essa pilha está vazia. Vimos pelas propriedades da pilha múltipla que uma pilha está vazia quando pilha.base[k] = pilha.topo[k] e, nesse caso temos um "underflow" local.

O processo de remoção propriamente dito, consiste em guardar o elemento referênciado pelo vector topo numa variável de auxiliar e, em seguida, subtrair uma unidade ao elemento do vector topo que faz referência a essa pilha.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Remover um elemento na k-ésima pilha
Parâmetro Entrada: Estrutura com NP pilhas e número da pilha a remover
Parâmetro de Saída: Estrutura com NP pilhas actualizada, Elemento removido
Retorno Função:Código de Erro ( Ok ou STACK_EMPTRY )
-------------------------------------------------------------------------------------------------*/
int desempilhar (TPMultipla pilha, int k, Item *x )
{
   if ( pilha.base[i] = pilha.topo[i] )
      return STACK_EMPTY;
   *x= pilha->item[ pilha->topo[i] ];
   pilha->topo[i] = pilha->topo[i] – 1;
   return OK;
{
```

A operação para inserir um elemento na k-ésima pilha é mais complexa. Ela consiste em primeiro lugar em verificar se essa pilha está cheia. Pelas propriedades da pilha múltipla sabemos que a k-ésima pilha está cheia quando pilha.topo[i] = pilha.base[i+1]-1, e nesse caso temos um "overflow" local.

O processo de inserção própriamente dito consiste em adicionar uma unidade ao elemento do vector topo que faz referência a essa pilha. Em seguida, inserir o elemento no vector pilha, cujo índice é referenciado pelo vector topo.

Mas, como tratar o overflow local? Uma possível estratégia consiste em encontrar uma pilha mais próxima à direita que tenha um espaço livre. Deslocar para à direita todos os elementos do vector que estão entre a pilha cheia e a pilha com espaço livre. Actualizar em seguida os índices dos vectores base e topo das pilhas movimentadas. A função que descrevemos a seguir, implementa essa estratégia.

```
/*-------------------------------------------------------------------------------------------------
Objectivo: Movimentar para a direita os campos de um conjunto de pilhas
Parâmetro Entrada: Estrutura com NP pilhas e número da pilha com overflow
Parâmetro de Saída: Estrutura com NP deslocada se for possível
Retorno Função:Código de Erro ( VERDADEIRO OU FALSO )
-------------------------------------------------------------------------------------------------*/
boolean paraDireita (TPMultipla *pilha, int k)
{
   int i;
   if ( (k < 1) || (k > NP-1) )
      return FALSE;
   if ( (pilha->topo[k] < pilha->base[k + 1]) )
     {
        for (i = pilha->topo[k] + 1; i > pilha->base[k]; i--)
              pilha->A[i] = pilha->A[i-1];
```

```
        pilha->topo[k]++;
        pilha->base[k]++;
        return OK;
      }
   return FALSE;
}
```

Mas se não for encontrado um espaço livre à direita, devemos procurar pela pilha mais próxima à esquerda com espaço livre e efectuar um procedimento equivalente. Deixamos como exercício a implementação de uma função que encontra esse espaço livre.

Estamos finalmente em condições de implementar uma função para inserir um novo elemento na k-éssima pilha que utiliza a estratégia anterior para tratar do overflow local.

```
/*-------------------------------------------------------------------------------------------------------
Objectivo: Inserir um elemento na k-éssima  pilha
Parâmetro Entrada: Estutura da pilha, número da pilha, elemento a inserir
Parâmetro de Saída: Estrutura da pilha actualizada
Retorno da Função: Código de erro ( STACK_FULL ou OK )
-------------------------------------------------------------------------------------------------------*/
int inserir (TPMultipla *pilha, int k, TItem x)
{
  int j;
  if ( (pilhaKcheia(*pilha, k)) && (k < NP-1) )
       // desloca p/direita todas as pilhas de [k+1..NP-1] em ordem reversa
       for( j = NP-1; j > k; j--)
           paraDireita(pilha, j);
 if  ( (pilhaKcheia(*pilha, k)) && (k > 0) )
      // desloca p/esquerda todas as pilhas de [1..k] (mas não a pilha 0)
      for ( j = 1; j <= k; j++)
           paraEsquerda(p, j);
 if ( pilhaKcheia(*p, k) )
    return STACK_FULL;
 pilha->topo[k]++;
 pilha->pilha[p->topo[k]]  = ch;
 return OK;
}
```

Observe que esta é apenas uma estratégia possível. Um procedimento mais eficiente poderia deslocar em primeiro lugar a pilha k+1 para direita. Se essa opção não fosse possível, deslocar apenas as pilha k-1 e a pilha k para à esquerda, e só no último caso é que iriamos deslocar as várias pilhas simultaneamente como na estratégia anterior.

5.6- Pilha Dinâmica

Uma forma mais eficiente de implementar esta estrutura, consiste em associar o ínicio da pilha a um ponteiro, denominado por topo. Com esta estratégia, as inserções e as remoções são feitas atravês desse ponteiro.

5.6.1- Estrutura de Dados

```
// Códigos de erro

#define NOT_FOUND       -1    // Item não existe
#define  OK              0    // Operação realizada com sucesso
#define  STACK_EMPTY     2    //  Pilha vazia
#define NO_SPACE         5    // Não há espaço de memória

// Estrutura de dados

typedef struct
 {
    int   chave;
    float  valor;
 }TInfo;

typedef struct apontador
 {
   TInfo  info;
   apontador  *prox;
 }TAtomo;

typedef struct
 {
     TAtomo * topo;
 } TPilhaDinamic;

typedey  enum   { FALSE = 0, TRUE = 1 } Boolean;
```

5.6.2- Implementação do Interface

```
/* ---------------------------------------------------------------------------------------------------
Especificação do Interface : pilhaDinamica.h
Objectivo: Disponibilizar as operações sobre uma pilha dinâmica
---------------------------------------------------------------------------------------------------*/
#ifndef  PILHADIM_H_INCLUDED
#define PILHADIM_H_INCLUDED

// Faça como exercício

#endif // PILHASEQ_H_INCLUDED
```

5.6.3- Implementação das Operações

De forma similar lista ligada, inicializar uma pilha, consiste em associa-la à um ponteiro com endereço nulo.

```
*/-------------------------------------------------------------------------------------------------------
Recebe:  Uma pilha
Objectivo: Inicializar uma pilha em alocação dinâmica
Devolve:  pilha dinâmica com ponteiros de controlo actualizados.
-------------------------------------------------------------------------------------------------------- */
void   iniciaPilhaDinamica ( TPilhaDinamic *pilha )
{
   pilha-> topo = NULL;
 }
```

Como a operações de empilhar e desempilhar já foram estudadas cuidaremos apenas das suas implementações.

```
*/-------------------------------------------------------------------------------------------------------
Recebe:  Uma pilha e um determinado registro
Objectivo: Inserir um átomo com essa informação no topo da pilha
Devolve:  Pilha actualizada e código de erro
-------------------------------------------------------------------------------------------------------- */
int  Empilhar ( TPilhaDinamic *pilha , TInfo x )
{
    TAtomo *pnovo = (TAtomo *) malloc(sizeof(TAtmo));
     if (pnovo == NULL)
         return  NO_SPACE;
     else {
              pnovo->info = x;
              pnovo->prox = pilha->topo;
              pilha->topo =  pnovo;
              return OK;
           }
}

*/-------------------------------------------------------------------------------------------------------
Recebe:  Uma pilha
Objectivo: Remover o átomo que está no topo da pilha
Devolve:  Pilha actualizada, informação removida e código de erro
-------------------------------------------------------------------------------------------------------- */
int  desempilhar ( TPilhaDinamica *pilha, TInfo *x )
{
   if ( vaziaPilhaDinamica(pilha) )
        return STACK_EMPTY;
   else {
              TAtomo  *pdel =  pilha->topo;
               *x  = pilha->info;
               pilha->topo =  pdel ->prox;
               free (pdel);
           }
 }
```

5.7- Exercícios

5.7.1- Considere 6 comboios numerados de 1 a 6 na entrada do estacionamento em forma de pilha. É poss´ıvel trocar a ordem dos comboios para 154632? E para 154623? Descreva a sequência de passos a ser realizada quando a troca puder ser feita.

5.7.2- Suponha que temos 4 registros com a seguinte ordem 1 2 3 4. Qual seria a sequência de operações de inserção (I) e remoção (R) em uma pilha em organização estática, para obter os registros na sequência 2 4 3 1? Por exemplo, se aplicar a sequência IIRIRR sobre a ordem inicial 1 2 3 obteriamos a sequência 2 3 1.

5.7.3- No caso de seis registros iniciais 1 2 3 4 5 6 seria possível obter a sequência 3 2 5 6 4 1? e a sequência 1 3 5 4 6 2 ?

5.7.4- Desenvolva uma função que recebe como parâmetro de entrada uma pilha armazenada num vector. Devolver essa pilha com os elementos ordenados na ordem crescente.

5.7.5- Desenvolva uma função que recebe como parâmetro de entrada uma cadeia de caracteres na forma XY, onde X é uma cadeia de caracteres formada por caracteres arbitrários e Y é o inverso de X. Utilize uma pilha para verificar se essa cadeira é de facto do tipo XY.

5.7.6- Dado um alfabeto formado pelas letras a, b e c, considere o seguinte conjunto de caracteres sobre esse alfabeto: c , aca, bcb, abcba, bbcbb, bacab, aabcbaa , ….. Qualquer cadeia deste conjunto tem a forma WcM, onde W é uma sequência de letras que só contêm a e b e M é o inverso de W. Desenvolva uma função que utiliza uma pilha para verificar se uma cadeia de caracteres recebida como parâmetro é do tipo WcM.

5.7.7- Desenvolva uma função que recebe como parâmetro de entrada uma lista sequencial armazenada num vector. Utilize obrigatóriamente uma pilha para inverter essa lista.

5.6.8- Desenvolva uma função que recebe como parâmetro de entrada uma pilha e um determinado valor k. Inserir um elemento na k_ésima posição da pilha.

5.7.9- Desenvolva uma função que recebe como parâmetro de entrada uma uma pilha armazenada num vector. Devolver uma cópia dessa pilha num outro vector.

5.7.10- Desenvolva uma função que recebe como parâmetro de entrada duas pilhas. Verificar se essas pilhas são iguais.

5.7.11- Desenvolva uma função que recebe como parâmetro de entrada uma pilha. Calcular o maior e o menor elemento dessa pilha.

5.7.12- Desenvolva um procedimento que recebe como parâmetro de entrada uma pilha. Imprima os seus elementos na mesma ordem do que foram inseridos.

5.7.13- Uma sequencia de operações E (Empilhar) e D (Desempilhar) numa pilha é válida se ela tem o mesmo número de Es e Ds. Formule uma regra que permita verificar se uma determinada operação é válida ou não.

5.6.14- Desenvolva uma função que recebe como parãmetro de entrada uma cadeia de caracteres composta por chavetas, parênteses e cochetes. Verificar se essa cadeia é bem formada. Por exemplo, as cadeias:

```
{[()]}

{ [({ } [ ] ( ) ) ] [ ] ( { ( ) } ) }
```

estão bem formadas ou passo que as cadeias:

```
{[(])}

{ [({ } [ ] ( ) ) ] [ ] ( { ( ) } ) ] }
```

não são.

5.7.15- Desenvolva uma função que recebe como parâmetro de entrada uma cadeia de caracteres terminada com um ponto. Utilize obrigatóriamente uma pilha para inverter cada palavra do texto, preservando a sua ordem. Por exemplo:

ESTE EXERCÍCIO É FÁCIL

A saída seria:

ETSE OICICREXE É OTIUM LICÁF

5.7.16- Considero que um estacionamento na rua direita de Luanda que é composto por uma única entrada e saída que guarda até dez carros. Para um cliente para retirar o seu carro, supondo que este não esteja próximo da saída, é necessário manobrar todos os carros que estão a bloquea-lo. O carro do cliente será manobrado para fora do estacionamento, e os outros carros voltarão a ocupar a mesma sequência inicial.
Desenvolva um programa para processar um conjunto finito de linhas de entrada. Cada linha de entrada contém um ‘E’, de entrada, ou um ‘S’ de saída, e o número da placa do carro. Assume-se que os carros cheguem e partam na mesma ordem que entraram no estacionamento. O programa deve imprimir uma mensagem sempre que um carro chegar ou sair. Quando um

carro chegar, a mensagem deve especificar se existe ou não vaga para o carro no estacionamento. Se não houver vaga, o carro partirá sem entrar no estacionamento. Quando um carro sair do estacionamento, a mensagem deverá incluir o número de vezes que o carro foi manobrado para fora do estacionamento para permitir que os outros carros pudessem sair.